Toward practical verification of outsourced computations using probabilistically checkable proofs (PCPs)

Srinath Setty Andrew Blumberg Michael Walfish
UTCS TR-10-26
The University of Texas at Austin
{srinath,mwalfish}@cs.utexas.edu blumberg@math.stanford.edu

Abstract

This paper describes preliminary results and outlines the plan of attack for an on-going project to develop a practical system for verifying outsourced computations. We want a client to be able to describe a computation to a server, get back a purported output and some auxiliary information, and use that auxiliary information to *verify* that the output is correct. For outsourcing to be worthwhile, the verification process should be substantially more efficient than simply executing the computation.

Our approach to this problem is to exploit the theory of probabilistically checkable proofs (PCPs). Specifically, our project seeks to build a bridge between the theory and an implementable system. We describe a protocol for outsourced computation that includes algorithmic refinements of the PCP protocol and end-to-end instantiation of the necessary steps (e.g., compilation to a form suitable for application of the PCP theorem). Although we are in the process of implementing the protocol and do not have experimental results, we present a detailed analysis that provides cause for optimism: our cycle and memory usage costs strongly suggest that our system will be useful. We focus on the example of matrix multiplication, where we show that for large matrices, our method achieves enormous savings for the client and requires feasible amounts of bandwidth.

1 Introduction and motivation

This paper describes preliminary results and outlines the plan of attack for an ongoing project to develop a practical system for verifying outsourced computations. Broadly speaking, we are interested in computations that are *too expensive* for the client to perform locally, and do not admit obvious procedures for verifying the correctness of a purported solution. Such computations range from complicated numerical algorithms operating on large matrices (which are polynomial but expensive) to NP-hard search problems where even the polynomial time check of the solution is too costly. Furthermore, we want to make only weak assumptions about the possible misbehavior of servers: we do not want to rely on replication methods but instead desire efficiently verifiable *proofs of correctness*.

Specifically, our goal is to realize the following high-level scheme, depicted in Fig. 1: a client sends a description of a computation, P, to a server (for example, in the form of a C program); the server executes P and returns the claimed output and some auxiliary information; the client uses the auxiliary information to verify that the output is correct; and the verification is substantially more efficient than simply executing the computation.

As a motivating scenario, consider a computationally limited device that wants to offload processing to the cloud [14]. For example, a smartphone might wish to outsource an expensive photographic manipulation for want of computational cycles [17]. However, the device owner may not be willing to assume the correctness of the cloud. As another scenario, some 30 projects (Seti@home, Folding@home, the Mersenne prime search, etc.) use the BOINC software platform [2, 3] to leverage the spare cycles of volunteers' computers to perform massive computations that would otherwise be infeasible. Unfortunately, a problem is that some "volunteers" run modified software that does not compute the answers correctly [4]. Today, these projects check volunteers' work by outsourcing the same computation to multiple hosts, but this approach does not protect against clients that are colluding or simply buggy. It would be far preferable if the central project computers could verify the correctness of a volunteer's purported answer.

Our approach to this problem is to exploit the remarkable work on the theory of probabilistically checkable proofs (PCPs) [7, 16, 20, 32]. The central theorem in the subject is that any language in $\mathcal{NP}$ admits proofs of membership that can be verified by checking a very small number of bits in the proof. Naturally, the fact that PCPs might provide a solution to the precise problem of verifiable outsourced computing has not been lost on the theorists working in the area.

However, as they appear in the theoretical literature, PCPs are not suitable for use in a real system for outsourcing: as we explain below in Section 3, the constants and overhead costs of certain aspects of the protocol make naive deployment infeasible. Broadly speaking, the central problem is the fact that the proofs are far too large to practically transmit or even for the verifier to instantiate. Indeed, the folklore is that PCPs are not really suitable for practical systems right now. As a consequence, there has been a flurry of work that addresses limited and specific classes of problems [5, 6, 21, 38–40] or makes strong trusted hardware assumptions [13, 22, 31, 35, 36].

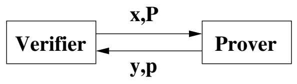

Figure 1—High-level depiction of verified outsourced computation. P is the computation, x is the input, y is the purported output, and p is auxiliary information.

In contrast, our point of departure is to carefully reexamine the grounds for this skepticism: we want to assess the feasibility of building a bridge between the theory and an implementable system for *general purpose* verifiable outsourced computation. We are not the first to revisit this issue: recently, there have been attempts to alleviate the costs of generic PCP [8, 20], with a view toward verifiable outsourcing. However, to date they have not been specified enough to be implementable.

By focusing on practical implementation from end to end, we have been forced to flesh out the details and grapple with the constraints imposed by a real system. Specifically, our project makes the following contributions:

1. **Design and specification.** In order to design a practical system, we have developed a number of refinements to the PCP protocol (discussed in more detail in Section 3). To be clear, we are not claiming substantial theoretical contributions. Rather, our innovation is in applying techniques (carefully constructed Merkle trees, batching, etc.) to reduce proof size and to amortize the cost of encoding problems in a format that is suitable for the theoretical machinery to apply. Using these refinements, we specify an end-to-end protocol for verifiable outsourcing.
2. **Feasibility analysis.** In Section 4, we perform a detailed analysis of our protocol in the context of matrix multiplication over arbitrary fields, a concrete problem that is a central primitive for machine learning and data mining applications, and an example of a numerical computation that BOINC distributes. Our results show that for large matrices, our method achieves enormous savings for the client and requires feasible amounts of bandwidth.
3. **Implementation roadmap.** As we are still implementing, we cannot present experimental results. However, in Section 5 we present a path toward fully implementing our approach, and extending it to a broader range of problems. This long-term program consists both of clearly achievable "get it working" problems, as well as research questions.

Thus, our results indicate that what seemed like a risky goal (at least to us) appears to be plausible. If we're right, then this opens the door to practical use of PCPs, which would be important for real networked systems, including classic distributed systems and data centers.

2 Related work

This section describes prior work on verifiable outsourced computations. We begin with work that shares our top-level summary: PCPs to verify computations.

PCPs to verify computations. Babai et al. give a protocol to verify computations in polylogarithmic time using PCPs [8], inspired by [11]. Goldwasser et al. [20] sketch a protocol based on interactive proofs to delegate computations; their scheme is asymptotically efficient for the prover and the verifier. Unfortunately, these papers do not specify the schemes sufficiently for their practical viability to be clear; notably, the constants that would be associated with actual reification of the protocols are unknown. Indeed, as mentioned in the introduction, one of our purposes here is to make the case for practical viability by carefully tracing through the required costs of a concrete and fully-specified protocol.

Secure multi-party computation. Another approach to verifiable outsourcing comes from secure multi-party computations. In such protocols, two (or more) mutually untrusting parties can compute an agreed-upon function of private data in a way that reveals publicly only the result, keeping the private data secret [10, 12, 19, 42]. By themselves, these protocols do not provide verifiable outsourced computations. However, Gennaro et al. [17] combine Yao's construction with Gentry's breakthrough results on homomorphic encryption [18] to provide verifiable non-interactive computing. Their construction is asymptotically efficient (in that it runs in polynomial time), but it inherits from Gentry's scheme, which is not yet practical to implement at scale on today's computers.

Even when considering only the secure multi-party protocols, the costs of expressing general computations in terms of Yao's garbled circuit constructions and the attendant oblivious transfer protocols are prohibitive. Indeed, the Fairplay system [9, 27] combines Yao's construction with innovative compilation techniques to transform programs written in a subset of C into secure multi-party computations. But Fairplay, although a technical tour de force, is essentially unusable for even relatively small problem instances. Although there have been subsequent refinements of Fairplay, the situation has not qualitatively improved (see [34] for a concrete discussion of the costs of Fairplay and its descendants in the context of computing the intersection of two sets).

Special-purpose protocols. As mentioned above, a number of works verify outsourced computations but are tailored to specific problem domains and encompass a circumscribed range of functions, e.g., database queries [39, 40], benchmarks [5], and linear algebra operations [6, 21]. In contrast, our ultimate goal is a protocol that supports functions encoded as C programs.

Trusted computing. Some work assumes trusted components on the prover: a trusted platform module (TPM), secure hardware token, hypervisor, or runtime platform [13, 22, 26, 29, 31, 35–37]. These assumptions do not hold in all environments; we want *any* computer to be able to be a prover.

Auditing. In some works, the verifier reruns the outsourced computation on a fraction of the input [31, 41]. This does not protect against an adversary that misreports a small number of strategic outputs and evades the audit. PCPs, in contrast, spread the entire output over the proof, so that even small deviations between the purported output and the correct answer break the proof. Another auditing approach is for the verifier to outsource the same computation to multiple machines [2]. However, this approach assumes that a majority of the provers are both bug-free and honest. We would prefer not to make such assumptions. Finally, Pioneer [38] audits an untrusted worker machine based on the time taken to perform a computation. This technique needs detailed knowledge of the worker's hardware; as mentioned above, we would like to avoid assumptions about the prover's platform.

3 Approach

In this section, we give an overview of our approach, and then discuss refinements. Our core PCP machinery follows a construction in [7]; the details of this construction are in Appendix C. Below, we use *verifier* to mean the entity that describes the computation and verifies the proof, and *prover* to mean the entity that executes the computation and issues the proof.

3.1 Overview

We begin with a seemingly trivial observation: if the prover gave the verifier a complete execution trace of the desired computation, including both the input and the output, and if the output in the trace were precisely the claimed output, then that execution trace would constitute a proof to the verifier that the computation was carried out correctly. Of course, verifying and receiving this execution trace would require more work than simply executing the computation. Thus, we need a way for the prover to issue a proof to the verifier but with the verifier able to check the proof by inspecting it in only a few places. This structure is precisely what PCPs enable.

We now give brief background on PCPs. Many of the PCP constructions in the literature (see [7, 32] and citations therein) concern 3-SAT: they allow a prover to prove that a given Boolean formula, ϕ, in 3-CNF has a satisfying assignment. (A 3-CNF Boolean formula contains True-or-False Boolean literals $b_1, \ldots, b_n$, and is a conjunction of clauses of the form, for example, $b_i \vee \bar{b_j} \vee b_k$. A satisfying assignment is a setting of each of the b_i to True or False such that every clause, and hence the entire formula, evaluates to True.) Of course, the assignment itself, $\vec{a}$, constitutes an (obvious) proof that ϕ has a satisfying assignment: the verifier could plug in $\vec{a}$ in every clause in ϕ.

The surprising content of the PCP theorem, however, is that in fact we can construct a compact proof that can be verified with arbitrarily high probability by inspecting a relatively small number of (randomly chosen) bits. In the case of 3-SAT, the PCP is a highly redundant *encoding* of $\vec{a}$ that spreads the information content of $\vec{a}$ over the entire proof. As a result, given ϕ, the verifier can check just a few bits in the PCP to be convinced (with a bounded error probability) that ϕ is satisfiable.

We now connect PCPs to the intuition above about execution traces. Consider the representation of P as a Boolean circuit, $\mathcal{C}_P$. A Boolean circuit is a set of interconnected gates, each of which has input wires and an output wire. P's input, x, will appear as values assigned to the input wires of some of the "early stage" gates of $\mathcal{C}_P$, and likewise P's output, y, will appear as the values of the output wires from "late stage" gates of $\mathcal{C}_P$. More generally, a True-False assignment to all of the wires in $\mathcal{C}_P$ is precisely an execution trace of P. Also, $\mathcal{C}_P$, a collection of Boolean literals, is equivalent to some Boolean formula, ϕ_P, in 3-CNF. Thus, *a satisfying assignment to ϕ_P, $\vec{a}_P$, is a valid execution trace of P.*[1] Moreover, $\vec{a}_P$ includes literals that correspond to x and y.

At this point, we are ready to apply PCPs: given ϕ_P, the prover issues a PCP that ϕ_P is satisfiable. Then the verifier carries out the following steps:

S1 The verifier inspects the PCP in a few places to check its validity; by the equivalence above, this establishes that the prover has a valid execution trace for P.

S2 The verifier must establish that the execution trace is based on the supplied input, x. But an assignment to the "input literals" is encoded in $\vec{a}_P$, and the verifier can (using a small number of random queries) check whether these literals match x.

S3 The verifier must now extract the output of the execution trace, y; this proceeds in the same fashion as the previous step.

(Note that this is a randomized protocol, and there is a probability of error in this process, but as usual it can be made arbitrarily small by repetition; see §4.)

The work of our project is to refine the approach in several ways, as described in the rest of this section:

- How can we avoid the entire (huge) proof passing from prover to verifier? (§3.2.)
- Given P, writing down ϕ_P is as expensive as simply executing P, which seems to defeat the purpose

[1]The observation that a program's execution can be encoded as a satisfiability instance is at the heart of Cook's proof that satisfiability is $\mathcal{NP}$-complete [15].

of outsourcing. Thus, how can we amortize and mitigate this cost, or at least avoid the *verifier* incurring it? (§3.3.)

- How can the verifier efficiently extract only the output literals from the proof? (§3.4.)
- How exactly can we go from a high-level description of a computation, P, into a 3-CNF representation, ϕ_P? Here, we do not innovate but just report our proposed approach, and why it ought to be practical. (§3.5.)

3.2 Reducing network costs

As so far described, the protocol requires the verifier to sink a proof that is polynomial in the number of literals of the Boolean formula. To drastically reduce network costs, the prover can commit to a *digest* of the proof. For each location in the proof that the verifier wants to inspect, it interactively queries the prover. The prover's responses must be consistent with the digest.

Although this high-level idea seems straightforward, implementing it efficiently in our context requires some care. Below, we describe four successive modifications, each reducing network costs.

First, as Kilian [24, 25] suggests, the prover encodes the PCP as a Merkle tree [30]; the *digest* is the tree's root. Specifically, each leaf in the tree is a collision-resistant hash of some portion of the proof and the interior nodes are collision-resistant hashes of their children. To inspect a location, l, in the PCP, the verifier submits l to the prover. The prover not only responds with the contents of l but also produces a *path* through the tree that proves that the contents were an input to the original digest. While this core idea is helpful, it was proposed in the context of generic PCP, rather than our specific scenario of verifying outsourced computation. In our scenario, steps S2 and S3 require the verifier to retrieve all of the input and output literals, which, naively, would require an interaction for each literal, which would be infeasible.

Thus, our second modification is to rearrange the Merkle tree so that the input literals are covered by a *single* leaf; we do likewise with the output literals. Now, retrieving the prover's claimed values for the input literals requires *one* invocation of the interactive protocol, and likewise retrieving y. To further reduce the costs, observe that the verifier begins with x, so step S2 consists only of the verifier checking that the hash of x was an input to the digest; x itself need no longer travel.

Third, observe that the verifier doesn't even *have* to have x in hand; all that step S2 *really* requires is knowledge of the hash of x. This observation offers significant network and memory gains to the verifier: now computations can be outsourced *even when the verifier does not know the input*, as might be the case for, say, a large data mining application. Of course, applying this insight requires a way for the verifier to receive the hash of the input from a source that it trusts, out of band.

Fourth, sometimes in step S1, the verifier randomly chooses to inspect the proof locations that contain the input literals. In those cases, given our current Merkle tree structure, the prover must send the full input to the verifier. This step could be costly. To make it cheaper, our final modification is to change the leaf node that covers the input literals from a flat hash of x to a Merkle tree encoding of x (and the out of band hash follows suit).

3.3 Amortizing setup costs

For any computation P, writing down a Boolean circuit equivalent to P, and hence writing down the 3-CNF Boolean formula ϕ_P, takes as much time as executing P and at least as much space (likely more). Thus, for outsourcing to be worthwhile, the verifier must be able to amortize these costs, and ideally avoid them altogether. Below we describe three ways that the verifier can do so.

First, if the verifier will verify the same computation with different inputs (as in the BOINC [2] examples given in the introduction), it can "reuse" ϕ_P, thereby amortizing the work to realize ϕ_P. This observation is not totally trivial: the reuse works because the PCP construction that we use (see Appendix C and [7]) allows the verifier to conduct verification based only on the PCP, on x, and on values that can be quickly computed from ϕ_P. (Simplifying slightly, these values are derived by encoding ϕ_P as a polynomial g_{ϕ_P}, choosing some random values $\{r\}$ and then returning $\{g_{\phi_P}(r)\}$.) Thus, with our protocol, the verifier would incur a time cost equivalent to executing P *once* but thereafter save work, since verification is far cheaper than executing P.

Second, we need to save the verifier space: ϕ_P is likely larger than the scratch space that the verifier would have needed to execute P. As an example, for $m \times m$ matrix multiplication, discussed in detail in §4.1, the size of ϕ_P is proportional to m^3 while the space to simply execute P is proportional to m^2.

To save the verifier space, we observe that the verifier doesn't need to retain ϕ_P, provided it has access to a small set $\{g_{\phi_P}(r)\}$, for random values of r; we call this set a *fingerprint* of ϕ_P. Thus, consider this scenario: a verifier (e.g., a BOINC project) pays the time cost *once* to realize ϕ_P. Then, the verifier pre-computes g_{ϕ_P} at every point in its domain (which costs time that is low-degree polynomial in the size of ϕ_P). The verifier then constructs a Merkle tree of these values, stores the value of the root node locally, and stores the pre-computed g_{ϕ_P} values in a cloud storage service [1]. Now the verifier can throw away ϕ_P, and verification requires little space: the verifier simply retrieves randomly selected g_{ϕ_P} values as needed. The verifier is protected against the storage service returning bogus values because the verifier knows the digest of the g_{ϕ_P} values.

Our third approach is to remove even the time and space cost of briefly materializing ϕ_P. The simple observation is that the verifier *never* needs to handle ϕ_P, if it can get a valid fingerprint from a source that it trusts (e.g., a registry mapping known P to pre-computed g_{ϕ_P}).

3.4 Extracting the inputs and outputs

A feature of the version of the PCP protocol that we use is that the input and output literals can be easily extracted from a particular location in the proof essentially in plaintext. In order to protect against a malicious prover, of course we cannot exploit this fact and must instead use a more involved protocol to coalesce the values of the input and output literals from random queries of the proof. However, for the purpose of the analysis in this paper, we assume a threat model in which the prover is *unreliable* but not malicious. That is, we assume that attacks in which the proof is generated correctly and is accepted by the verifier but has corrupted "naive" output literals do not occur. In work in progress, we are developing efficient schemes to remove this restriction (§5).

3.5 Converting programs into 3-CNF

So far, we have been assuming that *some* entity (the verifier or its delegate) can, given a description of P in a high-level language, realize ϕ_P, a Boolean formula in 3-CNF. We now discuss the methods by which this can be accomplished and the costs. We describe the work as being done by the verifier, even though, as mentioned above, the work might be done by a delegate.

At a high level, the main problem is to go from P to the circuit $\mathcal{C}_P$. (To go from $\mathcal{C}_P$ to ϕ_P is the easy part: by following established techniques [28], the verifier can transform $\mathcal{C}_P$ into a 3-CNF formula that has no more than five times as many clauses and five times as many literals as $\mathcal{C}_P$ has gates [28].)

To generate $\mathcal{C}_P$, one possibility is that the verifier can produce the circuit by hand. This is not as ridiculous as it sounds, especially for the types of linear algebra applications that are our initial target applications; indeed, in Section 4, we do precisely such a compilation by hand. It is tractable in part because the loop structure of matrix multiplication is so straightforward that unrolling it into a circuit is easy. More generally, there are many computations that can be expressed as Boolean circuits.

However, a major goal of our project is to outsource computations expressed in a restricted subset of a general-purpose programming language, like C. A starting point for our work (in progress) in this area is the innovative compiler module in the Fairplay [27] project; it uses SSA to produce efficient circuits from a subset of C. (Note that the tremendous inefficiencies in Fairplay come from the oblivious transfer protocol and consequent restrictions on optimization of the circuits.) We hope to concurrently refine our protocol and the compiler to extend our techniques to programs of reasonable size.

Metric	Count
Mult	$< \gamma((\frac{6}{\delta} + 48)\log^3 n + (\frac{6}{\delta} + 20)\log n + \frac{6}{\delta}\frac{\log n}{\log\log n} + 16)$
Add	$< \gamma(36\log^4 n + (\frac{6}{\delta} + 36)\log^2 n + \frac{6}{\delta}\frac{\log n}{\log\log n} + 20)$
Hash	$< \gamma(864\log^2 n + (\frac{180}{\delta} + 240)\log n + 12\frac{\log n}{\log\log n} + (\frac{12}{\delta} + 8))$
N/W	$< \gamma((432 + \frac{60}{\delta} + 864s)\log^2 n + (\frac{180s}{\delta} + 240s + 120)\log n + (\frac{48}{\delta} + 96)\frac{\log n}{\log\log n} + (\frac{96}{\delta} + 192)) + i + o$

Figure 2—Upper bounds on the verifier's cost—in terms of number of multiplication, addition, and collision-resistant hash operations, and bits transferred over the network between the verifier and the prover—for a Boolean formula with n literals. δ is set to 10^{-4} to get the error probability described in §4.1, s is the size in bits of a collision-resistant hash, and i and o are the respective sizes in bits of the input and output. The functions are polylogarithmic in the size of the computation; the work done by the verifier and the network resources consumed grow much slower than the size of the computation.

4 Analysis and suitability

This section gives a detailed analysis of the cost to the verifier of executing our protocol, focusing on the running example of matrix multiplication over a field. We answer two high level questions here: (1) What are the verification costs (§4.1)? (2) What class of computations are likely to result in a cheaper verification relative to just executing the computation locally (§4.2)?

4.1 Analysis

We first estimate the protocol's costs in terms of n, the number of literals in ϕ_P, and then apply these estimates to a specific computation and accompanying Boolean formula: matrix multiplication. We count the operations performed by the verifier and determine the network costs. (We do not include the costs of compiling or storing ϕ_P, which we expect to be amortized.) Exact counts are in Appendix D; simpler loose upper bounds are in Figure 2. *All of these costs are polylogarithmic in n, so grow much slower than the size of the computation.* The counts incorporate a term γ, which is specified so that the *error probability*—the probability that a correctly functioning verifier accepts an incorrect output—is at most $\frac{1}{2^\gamma}$. The counts also include hash operations, which come from the constructions described in §3.2.

We now estimate the cost of verifying the computation of "$m \times m$ matrix multiplication with 32-bit entries". Call this computation $\mathcal{M}$. To apply the counts above, we must determine n for the 3-CNF Boolean formula $\phi_{\mathcal{M}}$. We assume, naively, m^3 32-bit combinatorial multiplier circuits and $(m-1)m^2$ 32-bit adder circuits. As derived in Appendix A, a loose upper bound on n is $34784m^3 + 1120m^2(m-1)$. Taking this value of n in

		Computation (count)		N/W (MB)		Storage
m	Setup	Mult+Add	Hash (M)	Proof	I/O	Disk (MB)
10^3	B	2.00×10^9	-	-	-	11
10^3	O	0.26×10^9	47	986	11	11
10^4	B	2.00×10^{12}	-	-	-	1144
10^4	O	0.38×10^9	54	1153	1144	1144
10^5	B	2.00×10^{15}	-	-	-	114,440
10^5	O	0.49×10^9	62	1336	114,440	114,440
10^6	B	2.00×10^{18}	-	-	-	11,444,091
10^6	O	0.68×10^9	75	1661	11,444,091	11,444,091

Figure 3— Relative costs of performing matrix multiplication on $m \times m$ matrices. B is Baseline, O is our scheme, Mult is Multiplication, Add is addition, Hash is the number of hash operations in millions, I/O is input and output costs. Note that as the size of the input grows, the size of the proof sent over the network becomes insignificant relative to the input size.

the exact counts, we get the estimates in Figure 3, for different m. (We take $s = 160$ and $\gamma = 1$; to drive the error probability to $\frac{1}{2^\gamma}$, we must multiply the reported costs by γ). The figure also compares the costs to the baseline case of executing the computation locally. The savings in computation are enormous. There is some network cost relative to local computation, but, as the matrix size grows, this network cost is dwarfed by the network resources required to send the input and receive the output. Moreover, *any* outsourcing scheme would have to pay this input/output network cost.

4.2 Suitability

In our scheme, the cost of verification is polylogarithmic in the size of the computation (Figure 2) and the constants are small. Therefore, a verifier using our scheme will save work by outsourcing the computation when the local costs grow faster than this (e.g., the local costs are polynomial). We should note that in the simplified protocol we study here, we also depend on a compact representation as a Boolean circuit. As discussed in the next section, in future work we plan to relax this requirement.

However, there are many computations that meet these requirements already: various linear algebra operations, string pattern matching (as in a virus checker), context-free parsing, etc.

5 Research agenda

In this section, we outline our program for producing a practical system for verifiable outsourced computation.

Circuit generation. One of our goals is to work with arbitrary computations expressed in C, which requires making it feasible to compile such computations into concise formulas. §3.5 described our plans here.

Efficiency for the prover. We have so far focused on the verifier's efficiency. The computational burden on the prover is heavier. We are investigating protocol refinements to improve this.

Batching. If a verifier outsources multiple computations at once, then the verifier can save significant resources by *batching*: the prover generates a single proof for all the computations, instead of separate proofs for each. We are investigating modifying the verifier's protocol accordingly. In particular, the verifier should be able to use the formula or fingerprint of the single instance of the computation to obtain the fingerprint of the batch.

Using more efficient PCP constructions. There are constructions in the literature in which the verifier inspects $O(1)$ bits instead of $O(\log^4 n)$ [33] (where n is the number of literals in the Boolean formula).

A more expansive threat model. We are developing efficient ways to remove the assumptions we made about extracting the output literals from the proof (§3.4), allowing us to operate in a much broader threat regime.

A Estimates on the number of literals

Here we describe the steps we took to estimate the number of literals in a 32-bit adder and a 32-bit multiplier when they are represented in the form of a 3-CNF Boolean formula, as noted in §4.1. We examined an adder circuit from [23] that adds two 1-bit binary numbers. We then calculated the number of literals in that circuit after converting it to 3-CNF Boolean formula using the procedure from [28]. We found this count to be at most 35 literals. In order to get a 32-bit adder, a naive way is to use 32 1-bit ripple carry adders. Therefore, a 32-bit adder will have at most 1120 ($= 35 \times 32$) literals.

Next, we looked at a 4-bit multiplier circuit from [23] and estimated an upper bound on the number of AND gates and adders used by a 32-bit multiplier. A naive 32-bit multiplier will use at most $2\sum_{i=1}^{31} i$ adders and 2×32 AND gates. We then estimated the number of literals that would be present in the 3-CNF Boolean formula representation of a 32-bit multiplier. This count is at most 34784 ($= 2 \times 32 + (2\sum_{i=1}^{31} i) \times 35$).

B Background

Before presenting the core protocol, we describe some of the tools used by the PCP construction. We adapt the notation, terminology, and content here from [7]. Our protocol uses the PCP construction from [7] i.e., the construction of PCP($O(\log n)$, $O(\log^4 n)$) [7]. In this construction, the verifier uses $O(\log n)$ bits of randomness and examines $O(\log^4 n)$ bits of the proof, where n is the number of Boolean literals in the 3-CNF formula whose satisfiability is being proven. Our notation is summarized in Figure 4.

B.1 Arithmetization

Each clause in a Boolean formula, ϕ, in 3-CNF form, can be represented by a polynomial of degree 3 over a finite field. This can be done by replacing the Boolean

Symbol	Meaning
n	Number of Boolean variables (≥ 3)
q	a prime number ($\geq 100\lceil\log^4 n\rceil$)
F	finite field $\mathbb{Z}_q = \{0, ..., q-1\}$
H	subset $\{0, ..., \|H\|-1\}$ of F ($\|H\| = \lceil\log n\rceil$)
k	number of variables of polynomial ($\lceil\frac{\log n}{\log\log n}\rceil$)
$F_{d,k}$	set of $k-$variate polynomials of degree d over field F.
$\vec{\mathbf{h}}$	A vector, and h_i denotes the i^{th} component of $\vec{\mathbf{h}}$.

Figure 4—Notation used in the description of our scheme

$\vee$ with the field multiplication. Also, a Boolean variable, u, is replaced by $(1-x_u)$ and a negated variable, $\bar{u}$, is replaced by x_u, where x_u is a variable that can take values from the finite field.

Each clause of ϕ gets converted to one of the following polynomials depending on the number of negated variables. Let x, y, $z \in F$, a finite field. Now, define:

$$p_0(x,y,z) = (1-x)(1-y)(1-z)$$

$$p_1(x,y,z) = x(1-y)(1-z)$$

$$p_2(x,y,z) = xy(1-z)$$

$$p_3(x,y,z) = xyz$$

A clause is said to be type j, if the clause contains j negated variables. Assume that the negated variables always appear at the beginning of the clause, and the literals in the increasing order of their indices. The remaining variables appear after the negative variables but in the increasing order of their indices.

Let χ^j_ϕ, for $j = 0,1,2,3$ be the four clause-characteristic Boolean functions such that $\chi^j_\phi(i_1,i_2,i_3) = 1$, if and only if ϕ contains a clause of type j with variables u_{i_1}, u_{i_2}, and u_{i_3}.

Now if there exists a vector, $\vec{\mathbf{a}}$, then to check if $\vec{\mathbf{a}}$ satisfies the Boolean formula, ϕ, one has to check if the following four functions are identically zero for all $i_1, i_2, i_3 \in \{1, ..., n\}$ and $j = 0,1,2,3$:

$$f^j_\phi(i_1,i_2,i_3) = \chi^j_\phi(i_1,i_2,i_3)p_j(a_{i_1},a_{i_2},a_{i_3}) = 0$$

B.2 Zero-tester polynomials

The problem of verifying whether a polynomial is zero at every point in a finite field H^{3k} can be checked efficiently by using zero-tester polynomials [7]. This can be done by checking whether the polynomial in consideration multiplied by a zero-tester polynomial (chosen uniformly at random from a set of zero-testers) sums to zero in H^{3k}. Here we describe an example of a family of zero-testers that is used by our scheme.

For any $\vec{\mathbf{b}} \in F^{3k}$, define,

$$I_{b_i}(x_j) = \sum_{h\in H} {b_i}^h S^H_h(x_j)$$

$$R_{\vec{\mathbf{b}}}(x_1, ..., x_{3k}) = \Pi^{3k}_{i=1} I_{b_i}(x_i)$$

Now, the set of polynomials, $\cup_{\vec{\mathbf{b}}\in F^{3k}} R_{\vec{\mathbf{b}}}$, is a family of zero-testers in $F_{3k|H|,3k}$ [7].

B.3 Selector polynomials

Selector polynomials are useful for constructing low-degree extensions of functions. We define two of them here (an univariate and a multivariate polynomial). For any $w \in H$,

$$S^H_w(z) = \Pi_{y\in H; y\neq w}\left(\frac{z-y}{w-y}\right)$$

Note that $S^H_w(w) = 1$ and $S^H_w(x) = 0$ for any $x \in H$, and $x \neq w$.

For any $\vec{\mathbf{h}} \in H^k$,

$$S_{\vec{\mathbf{h}}}(\vec{\mathbf{x}}) = \Pi^k_{i=1} S^H_{h_i}(x_i)$$

Note that $S_{\vec{\mathbf{h}}}(\vec{\mathbf{h}}) = 1$ and $S_{\vec{\mathbf{h}}}(\vec{\mathbf{x}}) = 0$ for any $\vec{\mathbf{x}} \in H$, and $\vec{\mathbf{x}} \neq \vec{\mathbf{h}}$.

C Verified computation scheme

Here we provide a complete description of our protocol mentioned in §3. In our scheme, the outsourcing of a computation proceeds in two steps in the. In the first step, called the *compilation of the computation*, the computation in a high level language is converted into 3-CNF Boolean formula representation using a *compiler*. In the second step, called the *execution of the computation*, the computation expressed in 3-CNF Boolean formula is executed to generate the output.

In the protocol that we are going to describe, we assume that the verifier performs the compilation step and outsources the compiled Boolean formula to the prover. We begin by describing the computation performed by the prover. We then describe the algorithms used by the verifier to verify the proof generated by the prover.

C.1 The prover's protocol

The protocol that we describe here is for a correctly functioning prover. A malfunctioning prover can deviate from this protocol arbitrarily.

At a high level, the prover obtains the verifier's computation, ϕ, in the form of a 3-CNF Boolean formula. The verifier also specifies the input to the computation i.e., values for some of the variables in the Boolean formula. Let ϕ contain n variables. The prover first finds a satisfying assignment to ϕ, after assigning the verifier-supplied values to the corresponding variables of ϕ.

Once the prover finds the satisfying assignment, it encodes the satisfying assignment in a low-degree polynomial over a finite field. The prover also constructs other polynomials (partial-sum polynomials and a line table)

Metric	Count
Mult	$\leq \gamma(\frac{6(k\|H\|+1)}{\delta}\log(k\|H\|+1) + \frac{6k}{\delta} + 4((9k\|H\|+2)\log(3k\|H\|+1) + 3(k\|H\|+1)\log(k\|H\|+1)+4))$
Add	$\leq \gamma(36k^2\|H\|^2 + (\frac{6}{\delta}+36)k\|H\| + \frac{6}{\delta}k + 20)$
Hash	$< \gamma((\frac{18}{\delta}+24)k\log q + 72k^2\log q + 12k + (\frac{12}{\delta}+8))$
N/W	$< \gamma((\frac{18}{\delta}+24)ks\log q + (\frac{12}{\delta}+24)\log q + 72k^2 s\log q + \frac{6}{\delta}k\|H\|\log q + 36k^2\|H\|\log q + 12k\log q) + i + o$

Figure 5—Upper bounds on verifier's cost in terms of the count of computations: Mult (multiplication), Add (addition), Hash (collision-resistant hash), and N/W (bits transferred over the network between the verifier and the prover) for verifying an outsourced computation with a Boolean formula consisting of n literals. δ is set to 10^{-4} to get the error probability described in §4.1. The functions are polylogarithmic in the size of the computation; the work done by the verifier and the network resources consumed grow much slower than the size of the computation.

based on the assignment and the formula, ϕ. The prover evaluates these polynomials at all possible points in their domain. Once the prover evaluates these polynomials, it sends commitments about them to the verifier. This is done by constructing Merkle tree(s) whose leaf nodes contain the evaluated values and sending the root(s) of the Merkle tree(s) to the verifier.

Later, when the verifier starts the verification protocol, the prover has to return the requested leaf nodes from the Merkle tree(s). The prover also provides intermediate hashes in the Merkle tree starting from the requested leaf to the root of the Merkle tree. These hashes enable the verifier to check if the returned leaf node is part of the Merkle tree that the prover committed to. The commitments sent by the prover disallows the prover to change the proof generated in response to verifier's queries.

(1) Encoding the assignment vector as a low-degree polynomial. Let $\vec{\mathbf{a}}$ be an assignment vector. $\vec{\mathbf{a}}$ is a sequence of n bits. From the notation described in Figure 4 and the values specified for them, $|H^k| \geq n$. Therefore a $k-$tuple containing elements from H can uniquely identify a component in the assignment vector. Therefore the assignment vector, $\vec{\mathbf{a}}$ can be thought of as a function from H^k to $\{0,1\}$. Let this function be f_a.

The function f_a is then converted to a low-degree polynomial over a finite field, F by using the following set of transformations. (When a function from H^k to $\{0,1\}$ is transformed to a low-degree polynomial over a finite field, F, its a $k-$ variate polynomial with degree at most $k|H|$.) Let $p_f \in F_{k|H|,k}$ be the low-degree polynomial. For any $\vec{\mathbf{x}} \in F^k$,

$$p_f(\vec{\mathbf{x}}) = \sum_{\vec{\mathbf{h}} \in H^k} S_{\vec{\mathbf{h}}}(\vec{\mathbf{x}}) f_a(\vec{\mathbf{h}})$$

Since $p_f \in F_{k|H|,k}$, it can be represented by using q^k words each of length $\lceil \log q \rceil$ bits.

(2) Constructing the line table. The verifier would need some auxiliary information to use the low-degree encoding of the assignment vector. It is called a line table. A line table describes the "restriction" of the low-degree polynomial on all "lines" of F^k [7]. To construct the line table, the prover performs the following step: for each, $\vec{\mathbf{b}}$ and $\vec{\mathbf{s}} \in F^k$ and $x \in F$, the prover finds $(k|H|+1)$ coefficients of $P_{\vec{\mathbf{b}},\vec{\mathbf{s}}}$ where, $P_{\vec{\mathbf{b}},\vec{\mathbf{s}}}(x) = p_f(\vec{\mathbf{b}} + \vec{\mathbf{s}}x)$. Thus a line table is a function from F^{2k} to $F^{k|H|+1}$ (i.e., for all $\vec{\mathbf{b}}$ and $\vec{\mathbf{s}} \in F^k$, there is an entry in the line table). This line table can be represented by using q^{2k} words each of length $(k|H|+1)\lceil \log q \rceil$ bits.

(3) Constructing the partial-sum polynomial tables The prover also needs to provide more auxiliary information about the assignment vector and the computation. In particular, the auxiliary information enables the verifier to check if the satisfying assignment found by the prover actually satisfies the Boolean formula representation of the computation. To this aim, the prover arithmetizes the Boolean formula, as described in Appendix B.1.

Let f^j_ϕ be the Boolean function that describes ϕ (definition for f^j_ϕ can be found in Appendix B.1). It is a function from H^{3k} to $\{0,1\}$. Now, f^j_ϕ can be encoded as a low-degree polynomial in $F_{3k|H|,3k}$ similar to the low-degree extension of the assignment vector. Let $g^j_\phi \in F_{3k|H|,k}$ be the encoded polynomial. For $\vec{\mathbf{b}}$, $\vec{\mathbf{x}} \in F^{3k}$, let $r^j_{\vec{\mathbf{b}}}(\vec{\mathbf{x}}) = R_{\vec{\mathbf{b}}}(\vec{\mathbf{x}}) g^j_\phi(\vec{\mathbf{x}})$.

The goal is to construct polynomials that help the verifier to check if $\sum_{h \in H^{3k}} r^j_{\vec{\mathbf{b}}}(\vec{\mathbf{h}}) = 0$ for $j = 0,1,2,3$ and a randomly chosen $\vec{\mathbf{b}} \in F^{3k}$ (i.e., for a randomly chosen zero-tester polynomial). Note that this condition is equivalent to the condition for Boolean satisfiability described in Appendix B.1.

In order to enable efficient verification of the above condition by the verifier, the prover constructs partial-sum polynomials for every $\vec{\mathbf{b}} \in F^{3k}$, and $j = 0,1,2,3$. It is done using the following definition.

For $i = 1, ..., 3k$, the i^{th} partial-sum polynomial $g^j_{\vec{\mathbf{b}},i}$: $F^i \rightarrow F$, is defined as follows:

$$g^j_{\vec{\mathbf{b}},i}(x_1, .., x_i)$$

$$= \sum_{y_{i+1} \in H} \sum_{y_{i+2} \in H} ... \sum_{y_{3k} \in H} r^j_{\vec{\mathbf{b}}}(x_1, ..., x_i, y_{i+1}, y_{i+2} .., y_{3k})$$

Now, the prover creates four tables, one for each $j = 0,1,2,3$. Table T_j contains, for all $\vec{\mathbf{b}} \in F^{3k}$, for

Algorithm 1 Low-degree test (adapted from [7])

```
1. Input: A $k-$variate polynomial, $p_f$, and line-table, $T_a$.
2. Output: True if $p_f$ is $\delta-$close to a polynomial in $F_{k|H|,k}$
3. Set $\delta \leq 10^{-4}$
4. repeat
5.    Select $\vec{\mathbf{b}}$, $\vec{\mathbf{s}} \in F^k$ and $t \in F$ {define $P_{\vec{\mathbf{b}},\vec{\mathbf{s}}}(t) = \sum_{i=1}^{k|H|+1} t^{i-1} T_a(\vec{\mathbf{b}},\vec{\mathbf{s}})_i$, where $T_a(\vec{\mathbf{b}},\vec{\mathbf{s}})_i$ represents the $i^{th}$ element of the $(k|H|+1)-$dimensional vector, $T_a(\vec{\mathbf{b}},\vec{\mathbf{s}}) \in F^{k|H|+1}$}
6.    if ($P_{\vec{\mathbf{b}},\vec{\mathbf{s}}}(t) \neq p_f(\vec{\mathbf{b}} + \vec{\mathbf{s}}t)$) then
7.       return False
8.    end if
9. until $\lceil \frac{3}{\delta} \rceil$ times
10. return True
```

$i = 1, ..., 3k$, and for all $c_1, ..., c_{i-1} \in F$, the coefficients of the univariate polynomials $g^j_{\vec{\mathbf{b}},i}(c_1, ..., c_{i-1}, x)$.

C.2 The verifier's protocol

The verifier needs to know the 3-CNF Boolean formula, ϕ from which it can obtain low-degree extension of the function, χ^j_ϕ, for $j = 0, 1, 2, 3$ or it knows χ^j_ϕ, for $j = 0, 1, 2, 3$ from a trusted source.

At a high level, the verifier needs to perform the following steps:

V1 Check if the purported low-degree polynomial encoding of the assignment is actually a low-degree polynomial.

V2 Check if the purported assignment contained in the low-degree polynomial actually satisfies the Boolean formula representation of the computation.

V3 Check if the prover assigned the supplied input to the input variables

V4 Extract the output from the low-degree encoding of the assignment.

We now describes these high level steps in more detail.

(1) Low-degree test. First, the verifier checks if the purported low-degree polynomial constructed by the prover is indeed a low-degree polynomial of degree $k|H|$. This check is performed by the verifier by reading a constant number of words from the low-degree extension of the assignment and the line table (obtained by interactively querying the prover. The verifier also checks if the returned leaf nodes of the Merkle tree was present in the committed Merkle tree at the beginning of the verification protocol). Algorithm 1 sketches this test. Once this algorithm returns TRUE, the verifier is convinced that the polynomial constructed by the prover is $\delta-$close to a polynomial of degree $k|H|$.

Algorithm 2 Correcting a low-degree polynomial (adapted from [7])

```
1. Input: $\vec{\mathbf{x}} \in F^k$, a $k-$variate polynomial, $p_f$ ($p_f$ is $\delta-$close to a polynomial, $p \in F_{k|H|,k}$) and line-table, $T_a$.

2. Output: $p(\vec{\mathbf{x}})$
3. Choose a random $s \in F^k$
4. Choose a random $t \in F$
5. if ($P_{\vec{\mathbf{x}},\vec{\mathbf{s}}}(t) \neq p_f(\vec{\mathbf{x}} + \vec{\mathbf{s}}t)$) then
6.    return False
7. else
8.    return $P_{\vec{\mathbf{x}},\vec{\mathbf{s}}}(0)$
9. end if
```

Algorithm 3 Sum-check test (adapted from [7])

```
1. Input: $g^j_\phi \in F_{3k|H|,3k}$, $R_{\vec{\mathbf{b}}} \in F_{3k|H|,3k}$ and a table $T_j$ of partial-sum polynomials
2. Output: True if the product of $g^j_\phi$ and $R_{\vec{\mathbf{b}}}$ sum to 0 in $H^{3k}$.
3. Read the coefficients of $g^j_{\vec{\mathbf{b}},1}(x)$
4. if ($\sum_{x \in H} g^j_{\vec{\mathbf{b}},1}(x) \neq 0$) then
5.    return False
6. end if
7. Randomly choose $l_i \in F$ for $i = 1, ..., 3k$
8. for $i = 2$ to $3k$ do
9.    Read the coefficients of $g^j_{\vec{\mathbf{b}},i}(l_1, ..., l_{i-1}, x)$
10.   if ($\sum_{x \in H} g^j_{\vec{\mathbf{b}},i}(x) \neq g^j_{\vec{\mathbf{b}},(i-1)}(l_{i-1})$) then
11.      return False
12.   end if
13. end for
14. if ($R_{\vec{\mathbf{b}}}(l_1, ..., l_{3k}) \times g^j_\phi(l_1, ..., l_{3k}) \neq g^j_{\vec{\mathbf{b}},3k}(l_{3k})$) then
15.    return False
16. else
17.    return True
18. end if
```

(2) Sum-check test. Next, the verifier checks if the assignment encoded by the low-degree polynomial actually satisfies the Boolean formula, ϕ. To this aim, the verifier queries the partial-sum polynomial tables constructed by the prover. This check is performed by using the sum-check test.

In the sum-check test, the verifier first checks if the partial-sum polynomial tables constructed by the prover are "consistent" with one another at a randomly chosen location. Later, the verifier chooses a zero-tester, uniformly at random, from the family of zero-testers (the family that we described in Appendix B.2). Since the verifier either knows the Boolean formula or the low-degree extension of χ^j_ϕ, it can compute g^j_ϕ at any randomly chosen location (Definition of g^j_ϕ in Appendix C.1). This is done by using three words from the low-degree extension of the assignment and the value evaluated from χ_{ϕ}^{j} at the chosen random location.

The sum-check test checks if the product of the chosen zero-tester and g_{ϕ}^{j} at the chosen random location match the value generated by the prover. The sum-check test is performed once for each $j = 0, 1, 2, 3$, and is described in Algorithm 3. Once the sum-check test passes, the verifier is convinced that the assignment contained in the low-degree polynomial satisfies ϕ. (Since the verifier is querying a polynomial that is only $\delta-$close to the real polynomial, it applies the correcting procedure described in Algorithm 2 when reading from the low-degree polynomial.)

(3) Check input and extract output. Next, the verifier needs to check if the input variables of the purported assignment contain the verifier-supplied input. This can be done by querying the low-degree polynomial encoding of the assignment, and check if the input variables have the right values. To extract the output, the verifier queries the low-degree polynomial encoding of the purported assignment and extracts the values assigned to the output variables.

D Analysis

As noted in §4.1, we present a tight upper bound on the count of operations performed by a verifier for verifying a computation. These counts are expressed as functions of n, the number of literals in the 3-CNF Boolean formula representation of the computation. We stepped through each step of Algorithm 1, 2, 3, and counted the number of multiplications, additions, hash computations perfomed by the verifier. We also counted the number of bits transferred between the verifier and the prover for these Algorithms. These counts are function of γ, q, $|H|$, s, and k, which are in turn functions of n (as shown in Figure 4). Figure 5 shows these functions. Although not expressed directly in terms of n, they are all polylogarithmic in n.

Acknowledgments

We thank Marcos K. Aguilera, Hari Balakrishnan, Suman Jana, and Vitaly Shmatikov for helpful comments. This work was supported by ONR grant N00014-09-10757 and AFOSR grant FA9550-10-1-0073.

References

[1] Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3`.

[2] Berkeley Open Infrastructure for Network Computing (BOINC). `http://boinc.berkeley.edu`.

[3] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID*, 2004.

[4] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *CACM*, 45(11):56–61, Nov. 2002.

[5] S. Ar and J.-Y. Cai. Reliable benchmarks using numerical instability. In *SODA*, 1994.

[6] M. J. Atallah and K. B. Frikken. Securely outsourcing linear algebra computations. In *ASIACCS*, 2010.

[7] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, Berlin, Germany, 1999.

[8] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking Computations in Polylogarithmic Time. In *STOC*, 1991.

[9] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP A System for Secure Multi-Party Computation. In *CCS*, 2008.

[10] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *STOC*, 1988.

[11] M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.

[12] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *STOC*, 1988.

[13] A. Chiesa and E. Tromer. Proof-Carrying Data and Hearsay Arguments from Signature Cards. In *ICS*, 2010.

[14] B.-G. Chun and P. Maniatis. Augmented smart phone applications through clone cloud execution. In *HotOS*, 2009.

[15] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, 1971.

[16] I. Dinur. The PCP theorem by gap amplification. *J. ACM*, 54(3):12, 2007.

[17] R. Gennaro, C. Gentry, and B. Parno. Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proc. CRYPTO*, 2010.

[18] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[19] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *STOC*, 1987.

[20] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating Computation: Interactive Proofs for Muggles. In *STOC*, 2008.

[21] J. Groth. Linear Algebra with Sub-linear Zero-Knowledge Arguments. In *CRYPTO*, 2009.

[22] A. Haeberlen. A Case for the Accountable Cloud. In *LADIS*, 2009.

[23] R. H. Katz. *Contemporary logic design*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[24] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, 1992.

[25] J. Kilian. Improved Efficient Arguments (Preliminary Version). In *CRYPTO*, 1995.

[26] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *SOSP*, 2009.

[27] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-Party Computation System. In *USENIX*

Security, 2004.

[28] J. P. Marques-Silva and K. A. Sakallah. Boolean Satisfiability in Electronic Design Automation. In *DAC*, 2000.

[29] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Eurosys*, 2008.

[30] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Proc. CRYPTO*, 1987.

[31] F. Monrose, P. Wycko, and A. D. Rubin. Distributed Execution with Remote Audit. In *NDSS*, 1999.

[32] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 2007.

[33] A. Polishchuk and D. A. Spielman. Nearly-linear Size Holographic Proofs. In *STOC*, 1994.

[34] R. A. Popa, H. Balakrishnan, and A. Blumberg. VPriv: Protecting Privacy in Location-Based Vehicular Services. In *USENIX Security*, 2009.

[35] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-Based Cloud Computing. In *TRUST*, volume 6101 of *LNCS*, pages 417–429, 2010.

[36] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *HotCloud*, 2009.

[37] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[38] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. In *SOSP*, 2005.

[39] R. Sion. Query Execution Assurance for Outsourced Databases. In *VLDB*, 2005.

[40] B. Thompson, S. Haber, W. G. Horne, T. Sander, and D. Yao. *Privacy-Preserving Computation and Verification of Aggregate Queries on Outsourced Databases*. Springer-Verlag, 2009.

[41] T. Wang and L. Liu. Execution Assurance for Massive Computing Tasks. *IEICE Transactions on Information and Systems*, E93-D(6), 2010.

[42] A. C.-C. Yao. How to Generate and Exchange Secrets. In *FOCS*, 1986.